Confluent Orthogonal Drawings of Syntax Diagrams

Michael J. Bannister, David A. Brown, and David Eppstein

Department of Computer Science, University of California, Irvine*

Abstract. We provide a pipeline for generating syntax diagrams (also called railroad diagrams) from context free grammars. Syntax diagrams are a graphical representation of a context free language, which we formalize abstractly as a set of mutually recursive nondeterministic finite automata and draw by combining elements from the confluent drawing, layered drawing, and smooth orthogonal drawing styles. Within our pipeline we introduce several heuristics that modify the grammar but preserve the language, improving the aesthetics of the final drawing.

1 Introduction

The languages of computing, such as programming languages and data exchange formats, are typically specified using a finite set of rules called a grammar, and these rules are usually given in Backus–Naur Form or one of its extensions. Backus–Naur Form provides a notation rich enough to express all context-free grammars, and in turn most grammars of practical interest, while being easily machine readable. However, being a purely textual representation, it is perhaps less readable by humans. For this reason, Jensen and Wirth used a graphical representation of context-free grammars, called syntax diagrams, when defining the programming language Pascal [1].[1] We investigate the problem of generating syntax diagrams for context-free grammars and provide several heuristics optimizing the aesthetics of the resulting drawing. Our work provides the first algorithmic study of this problem and the first system that attempts to optimize the resulting diagram for readability rather than directly translating a given grammar into a diagram.

Recall that a *context-free grammar* is defined by four values N, Σ, R, S. In this 4-tuple, N is a set of *nonterminal symbols*, Σ is a set of *terminal symbols*, R is a set of *production rules* of the form $A \to \beta$ where A is a nonterminal symbol and β is a (possibly empty) string of terminal and nonterminal symbols, and S is a nonterminal symbol designated as the *start symbol*. A string σ of terminal symbols belongs to the language defined by the grammar when there exists a sequence of *rewrite steps* starting from S and ending at σ, each of which replaces

* Michael Bannister and David Eppstein were supported in part by NSF grant CCF-1228639.

[1] Jensen and Wirth were not the first to use syntax diagrams [2], but they popularized them, and these diagrams have been widely used since.

$$
\begin{aligned}
\langle \text{S-expression} \rangle &\to \langle \text{atomic symbol} \rangle \\
&\quad \mid (\langle \text{S-expression} \rangle . \langle \text{S-expression} \rangle) \\
&\quad \mid (\langle \text{S-expression list} \rangle) \\
\langle \text{S-expression list} \rangle &\to \epsilon \mid \langle \text{S-expression} \rangle \langle \text{S-expression list} \rangle \\
\langle \text{atomic-symbol} \rangle &\to \langle \text{LETTER} \rangle \langle \text{atom part} \rangle \\
\langle \text{atom part} \rangle &\to \epsilon \mid \langle \text{LETTER} \rangle \langle \text{atom part} \rangle \mid \langle \text{number} \rangle \langle \text{atom part} \rangle \\
\langle \text{LETTER} \rangle &\to \text{A} \mid \text{B} \mid \text{C} \mid \cdots \mid \text{Z} \\
\langle \text{number} \rangle &\to 0 \mid 1 \mid 2 \mid \cdots \mid 9
\end{aligned}
$$

Table 1. A context-free grammar for the language of S-expressions in LISP 1.5 [3].

a nonterminal symbol A in the current string with a string β such that $A \to \beta$ is a production rule in the grammar. Table 1 gives an example grammar for the S-expressions in the programming language LISP 1.5.

A *regular grammar* is one in which the production rules all have the form $A \to b$, $A \to bC$ or $A \to \epsilon$, where A and C are nonterminals, b is a terminal, and ϵ is the empty string. An example of a regular grammar is the part of the LISP 1.5 grammar defining $\langle$atom part$\rangle$. Languages definable by regular grammars are exactly the regular languages, whose equivalent characterizations include being recognizable by nondeterministic finite automata (NFAs). For these languages, we could use graph drawings of an NFA state graph as a graphical representation, by drawing an *st*-digraph with edges labeled by terminal symbols. A string σ is in the language if and only if there is a directed path through the graph from s to t such that the concatenation of the edge labels is equal to σ. Unfortunately, such a representation will not work for non-regular languages.

To graphically represent context-free languages we turn to syntax diagrams. Although other authors used syntax diagrams earlier [2], they were popularized by the Pascal User Manual and Report by Jensen and Wirth [1]. The style has been praised for its readability [4] and pedagogical value [5], and has been used by the Smalltalk-80 Blue Book [6], JSON Data Interchange Standard [7], and the W3C technical report on CSS [8]. Several software packages have been created to automate the drawing of syntax diagrams [9–11]. These software packages provide little to no optimization of the drawing, providing only a one-to-one translation of the Extended Backus–Naur grammars into syntax diagrams. Until now, there does not seem to be any algorithmic research involving the generation and optimization of syntax diagrams.

We introduce a new formalization for syntax diagrams consisting of a collection of *st*-digraphs (see e.g., Figure 3), each representing the possible expansions of a single nonterminal symbol, with each edge in each graph labeled by either a terminal or a nonterminal symbol. As before a string is in the language if and only if the string can be represented by a directed path from s to t in the start symbol's *st*-digraph. However, when this path would contain a nonterminal symbol, we

recurse into the *st*-digraph corresponding to that symbol. The concatenation of the terminal symbols in the resulting system of recursively generated paths should match the sequence of terminal symbols in the given string.

Without further optimization this formalization merely gives a new notation for writing production rules, but it has two advantages over extended BNF. Firstly, it gives us additional freedom in our representation: a BNF grammar can only describe syntax diagrams formed by a collection of disjoint paths between the two terminals, and extended BNF can still only describe syntax diagrams in the form of series-parallel graphs, while our diagrams are not restricted in these ways. Secondly, as we describe below, we can use this notation to directly represent the junctions and tracks of a confluent drawing style [12], in which a path through the graph is only valid if it is a smooth path, such as in Figure 1 (right). It is this drawing style that gives rise to the occasionally used alternative name "railroad diagrams" for syntax diagrams.

Our drawings will combine confluent drawing with Sugiyama-style layered drawing [13, 14] using smooth orthogonal edge shapes [15]. The combination of confluent and layered drawing has been studied before [16], but in a different way. Past work considered confluent drawing as a technique for visualizing a specific graph, and involved a search for subgraphs that could be more concisely expressed using confluence. In our application, the graph (NFA) representation that we work with already encodes the confluent features of the drawing: its vertices become confluent junctions in the drawing, and its edges become the boxes and connecting segments of track of the drawing (Figure 9). Rather than searching for graph features that can become confluent, our focus is on modifying the underlying NFA to produce a simpler and higher-quality drawing while preserving the equivalence of the underlying context-free language described by the drawing.

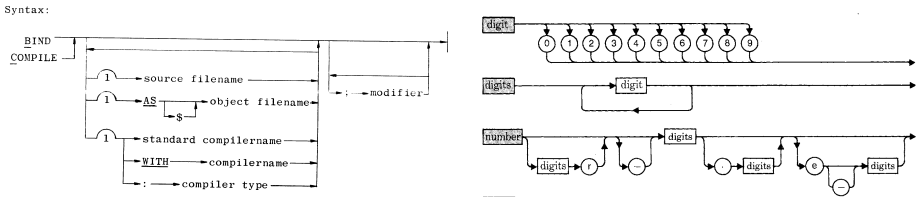

Fig. 1. A syntax diagram from the CANDE Information Manual (left) and a confluent syntax diagram from the Pascal User Manual and Report (right).

1.1 Software pipeline

We describe our method for producing syntax diagrams with the framework of a generic software pipeline. In the first step of our pipeline, we convert the grammar to our internal representation, which we will call the *NFA representation*. This

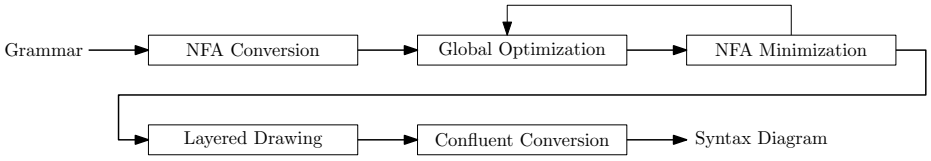

Fig. 2. A flow chart describing our software pipeline.

representation consists of a family of *st*-digraphs, initially one for each nonterminal symbol, whose edges are labeled by (terminal and nonterminal) symbols in the grammar or ϵ (the empty string). To construct the *st*-digraph for the nonterminal symbol A we convert each production of the form $A \to B_0 B_1 \cdots B_{r-1}$ into a directed path of length r labeled by the symbols B_0, B_1, B_{r-1}. Then all of the beginning and ending vertices are respectively merged together. Finally, we add to the graph two extra ϵ-labeled edges, one at the beginning and one at the end. See Figure 3 for the complete NFA representation of LISP 1.5.

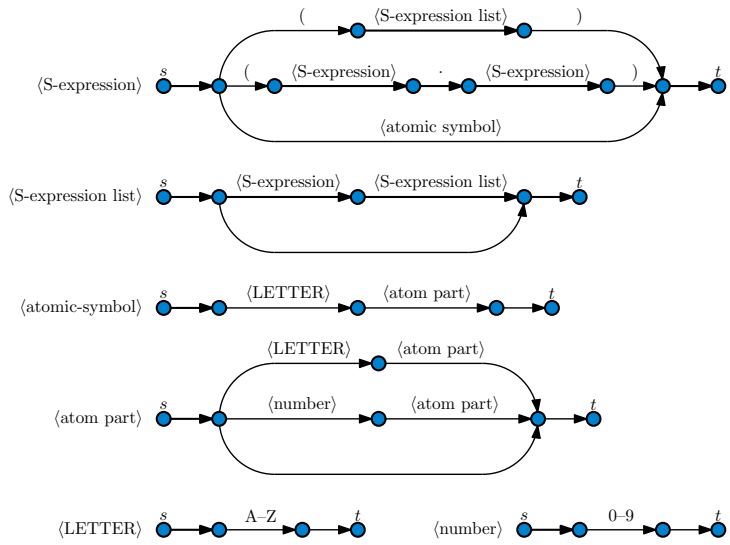

Fig. 3. The initial NFA-representation of S-expressions in the LISP 1.5 grammar.

The second and third steps in the pipeline attempt to reduce the number of total symbols in the NFA representation, through both global optimizations that act on the entire system of graphs and local optimizations that act on a single graph. The local optimization part of the pipeline is a form of the well-studied problem of NFA minimization. In general exact NFA minimization is PSPACE-hard [17, 18], and furthermore approximating the minimum NFA efficiently to within an $o(n)$ approximation ratio is also PSPACE-hard [19]. However, since the

problem is of practical importance there are many heuristic approaches [20, 21]. In this paper, we use simple heuristics motivated by the structure of real-world grammars and typical simplifications found in hand drawn syntax diagrams, rather than attempting to implement the more complex heuristics devised for minimizing NFAs without regard to their appearance as a diagram.

Once the NFA representation is optimized, we draw each of the *st*-digraphs in a layered Sugiyama style [13, 14], rotated horizontally to direct edges from left to right. In these graphs, the only directed cycles come from tail recursion elimination, so rather than searching for a small feedback arc set to determine the reversed edges in the drawing, we maintain such a set during the process of NFA minimization and add to it whenever we perform a tail recursion elimination step. In this way, we can ensure that all the tokens in the drawing are traversed from left to right. Standard layered drawing optimizations are applicable in this stage, but were not implemented in our experiments as we were primarily interested in optimizing the NFA representation. Finally, we convert the layered drawing into a confluent syntax diagram.

1.2 Contributions

Our contributions in this paper are summarized below.

- We formalize an abstract representation of syntax diagrams as a collection of mutually recursive NFAs, allowing the application of NFA minimization heuristics beyond what is possible with EBNF.
- We formulate a software pipeline for producing syntax diagrams, based on NFA minimization and confluent layered graph drawing.
- We develop a family of fast and simple NFA minimization heuristics, together with global heuristics that recombine multiple NFAs.
- We describe a geometric layout method based on a horizontal Sugiyama layered drawing, where we reinterpret the vertices and edges in a layered drawing of an NFA as the junctions and vertices of a confluent drawing.
- We provide a proof-of-concept implementation that produces human quality syntax diagrams for real-world context-free languages.
- Finally, we experimentally evaluate the quality of our heuristics.

2 Global minimization heuristics

A *global minimization heuristic* seeks to minimize the total number of labeled edges in an NFA representation via the modification of two or more of the *st*-digraphs in the representation. The only global heuristic that we consider is *nonterminal nesting*, in which a single nonterminal edge in one graph is replaced by the entire graph corresponding to that nonterminal edge. Since the goal is to reduce the total number of symbols in the NFA representation, we enforce the following restrictions when nesting a graph H (corresponding to a nonterminal A) into another graph G:

– A cannot be the start symbol.
– G and H must be two distinct graphs.
– If H has more than one non-ϵ edge, then A must occur only once in the whole system of digraphs, and its occurrence must be in G.
– The number of symbols in the graph produced by nesting H into G must be less than a predefined threshold k.

The final restriction above is intended to keep the size of each individual st-digraph to a human-readable level. The nesting heuristic can be seen to have been used in some hand-drawn syntax diagrams (e.g., the JSON syntax diagrams), but it does not appear to be used by previous syntax diagram software. See Figure 4 for an example of nesting with the LISP 1.5 grammar.

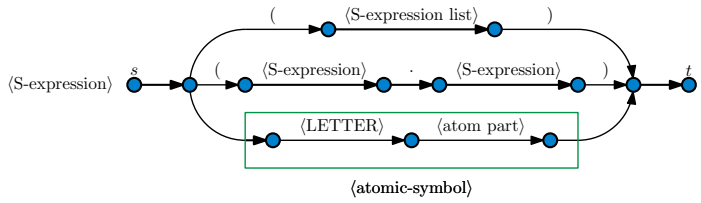

Fig. 4. An example of nesting the ⟨atomic symbol⟩ st-digraph into the ⟨S-expression⟩ st-digraph, within the LISP 1.5 grammar.

3 Local minimization heuristics

A *local minimization heuristic* seeks to minimize the total number of labeled edges in a single st-digraph within the NFA representation. Many of these optimizations can be seen in hand-drawn syntax diagrams.

3.1 Tail recursion loop back

The st-digraphs produced from a grammar, before optimization, are acyclic, and nesting preserves acyclicity. However, hand-drawn syntax diagrams typically contain cycles, which we introduce as a replacement for tail-recursive grammars using the *loop back heuristic*. If a nonterminal A appears exactly once in its own st-digraph and the edge on which it appears has t' (the only incoming neighbor of t) as its destination, then we change the destination of the A-labeled edge from t' to s' (the only outgoing neighbor of s) and we change its label from A to ϵ. Although this does not reduce the number of edges in the st-digraph, it does reduce the number of labeled edges and improves the readability of the drawing. In addition, by reducing the number of occurrences of A as a label, it may cause nesting operations to become possible that were previously forbidden. The edges that are modified by this heuristic will be the only ones directed backwards in our eventual drawings. See Figure 5 for an example of this construction.

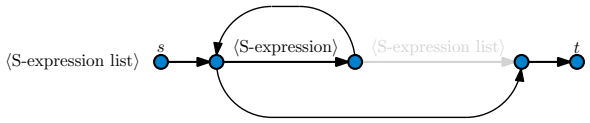

Fig. 5. An example of tail recursion loop back of ⟨S-expression list⟩ in the LISP 1.5 grammar. The removed edge has been colored gray.

3.2 Parallel state elimination with squish heuristic

The *squish forward* heuristic is used to reduce the number of nonempty symbols when there are parallel occurrences of the same symbol. If two edges $e_1 = (u, v_1)$ and $e_2 = (u, v_2)$ are labeled by the same symbol $A \neq \epsilon$, then we replace e_1 and e_2 with $f = (u, t)$ labeled A, $f_1 = (t, v_1)$ labeled ϵ and $f_2 = (t, v_2)$ labeled ϵ. We similarly define the *squish backward* heuristic, to be the squish forward heuristic applied to an st-digraph in which all of the edges have been reversed. See Figure 6 for an example of this heuristic.

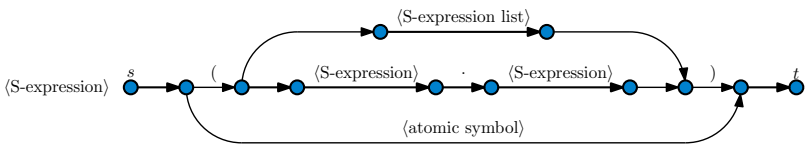

Fig. 6. An example of the squish heuristics applied to ⟨S-expression⟩ in the LISP 1.5 grammar. The squish forward combines the open parenthesis and the squish backward combines the closing parenthesis.

3.3 Epsilon transition removal

Our previous optimizations may introduce ϵ-labeled edges. We attempt to remove redundant ϵ-edges using the *epsilon removal* heuristic. If $e = (u, v)$, with $u \neq s$ and $v \neq t$, is an ϵ labeled edge, such that e is not a reversed edge (introduced via the loop back heuristic), and either e is the only outgoing edge of u or the only incoming edge to v, then the edge e is removed by merging u and v. We iteratively find and remove such edges until no such edge exists.

3.4 Confluent pinch

Our final local optimization would not qualify as an NFA optimization, as it does not attempt to reduce the number of symbols. Instead, the *confluent pinch* heuristic attempts to reduce crossings in the final drawing by removing directed complete bipartite subgraphs (which can be created by the squish heuristic), replacing each one by a single "crossing" vertex. If a digraph contains a set of

vertices U and a set of vertices V such that there is an ϵ labeled edge (u, v) for all $u \in U$ and $v \in V$, then we remove all such edges and add ϵ-labeled edges (u, w) for all $u \in U$ and (w, v) for all $v \in V$.

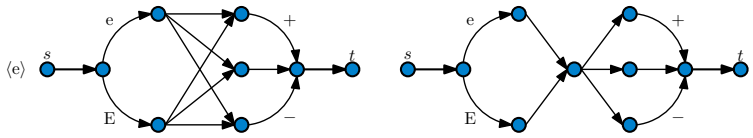

Fig. 7. An example of confluent pinch for scientific notation in the JSON grammar.

3.5 Implementing the heuristics

The application of one heuristic may create new optimization opportunities with respect to a previously applied heuristic. Therefore, we perform multiple rounds of optimization, applying all possible heuristics within each round, until no further optimizations are possible or a maximum number of rounds have been completed. In Figure 8 we see the optimized NFA representation of S-expressions in LISP 1.5, as produced by our implementation of these heuristics.

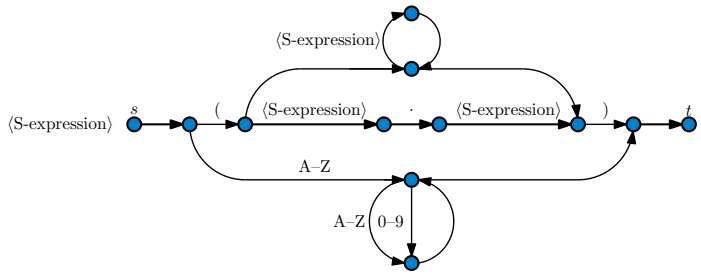

Fig. 8. Optimized NFA representation for S-expressions in LISP 1.5.

4 Sugiyama layering

Once the NFA representation has been minimized, we give each of the *st*-digraphs a Sugiyama-style layered drawing, using the standard layered-drawing pipeline for layout and crossing minimization. One modification that we make to this pipeline is that it is neither necessary nor desirable to compute a feedback arc set of the *st*-digraphs. Instead, the set of edges introduced during the loop back heuristic already form a feedback arc set with edges which should loop back into

the drawing. Since we are using an orthogonal drawing style, we add bends to edges to allow them to shift their vertical positions from one layer to the next, and use an interval-graph coloring algorithm to place the vertical connectors of these bent edges into a small number of columns.

In the final step of our algorithm, we reinterpret the vertices and edges in the resulting orthogonal drawing as the confluent junctions, track segments, and vertices of a confluent drawing. We place a vertex of the confluent drawing at the middle of each edge of the layered drawing whose label is not ϵ, with the confluent vertex being given the same label as the *st*-digraph edge label. We place a confluent junction at each vertex of the layered drawing, connected to a segment of confluent track for each incident edge of the layered drawing. Additionally, confluent junctions are created by the overlapping of edges with a common source. The orientation of the track at each confluent junction is determined by two factors: whether it connects to an earlier or a later layer, and whether it is a forward or reversed edge in the layered drawing. The result of this conversion step is our final syntax diagram. See Figure 9 for an example of this final conversion step.

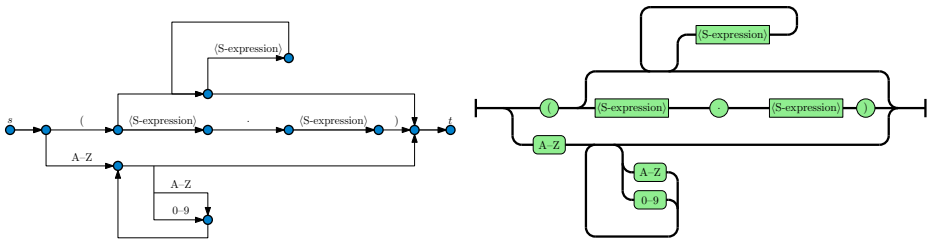

Fig. 9. The final confluent conversion from an orthogonal layered drawing into a syntax diagram for LISP 1.5 S-expressions.

5 Experimental results

In order to validate the heuristic optimizations performed by our implementation, we tested them on a set of eight real-world context-free grammars collected by Neal Wagner at the web site `http://www.cs.utsa.edu/~wagner/CS3723/grammar/examples2.html` together with the Lisp 1.5 and JSON grammars. For each grammar, we measured the area of our drawing (in units of rows and columns), the number of tokens (boxes) in the drawing, and the total number of connected components, both before and after optimization. The results are shown in Table 2.

As these results show, our optimizations were not always effective at reducing the total area of our drawings, and in some cases even increased the area. However, we typically achieved more significant reductions in the numbers of tokens and

Name	optimized?	area	tokens	components
Canadian post codes (simple)	unoptimized	17	6	1
	optimized	17	6	1
Canadian post codes (complex)	unoptimized	693	69	9
	optimized	1121	65	5
Ottawa course codes	unoptimized	520	46	15
	optimized	570	36	5
Palindromes	unoptimized	583	105	2
	optimized	583	105	2
Nonempty data files (repetitive)	unoptimized	182	22	8
	optimized	132	11	3
Nonempty data files (recursive)	unoptimized	143	22	7
	optimized	130	7	1
Pascal variable declarations	unoptimized	156	21	7
	optimized	247	12	3
Pascal type declarations	unoptimized	475	52	16
	optimized	486	30	6
LISP 1.5	unoptimized	165	19	6
	optimized	105	9	1
JSON	unoptimized	539	90	15
	optimized	651	42	5

Table 2. Experimental results

connected components of the drawings, which we believe to be helpful in reducing their visual clutter. Additionally, it can be seen that our optimizations are typically more effective on grammars with larger numbers of nonterminals, and less effective on grammars that have only a very small number of nonterminals, because in those cases no nesting will be possible.

We did not directly compare the results of other available syntax diagram drawing systems, but the ones we tested all appear to translate the input grammar to a diagram directly, without optimization; therefore, we believe that the results of testing them would be similar to the unoptimized lines of the table.

6 Gallery of examples

We present in Figure 10 and Figure 11 two complete examples of syntax diagrams of real-world grammars drawn by our implementation. For the LISP 1.5 grammar, our optimizations reduce the entire grammar to a single graph. We also present our results for the JSON grammar, which we believe (despite its obvious flaws) compares favorably with the official hand-drawn JSON syntax diagrams. Note in particular that the JSON ⟨number⟩ subgraph is not series-parallel, and therefore could not be represented by EBNF.

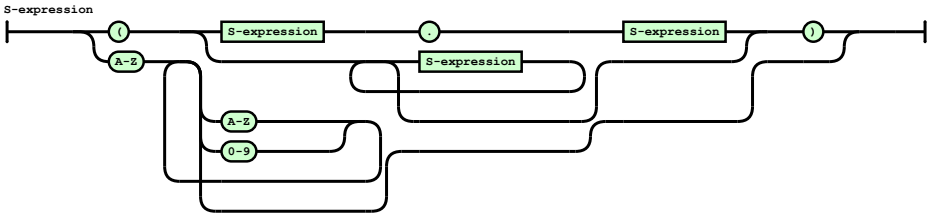

Fig. 10. A syntax diagram for S-expressions in LISP 1.5.

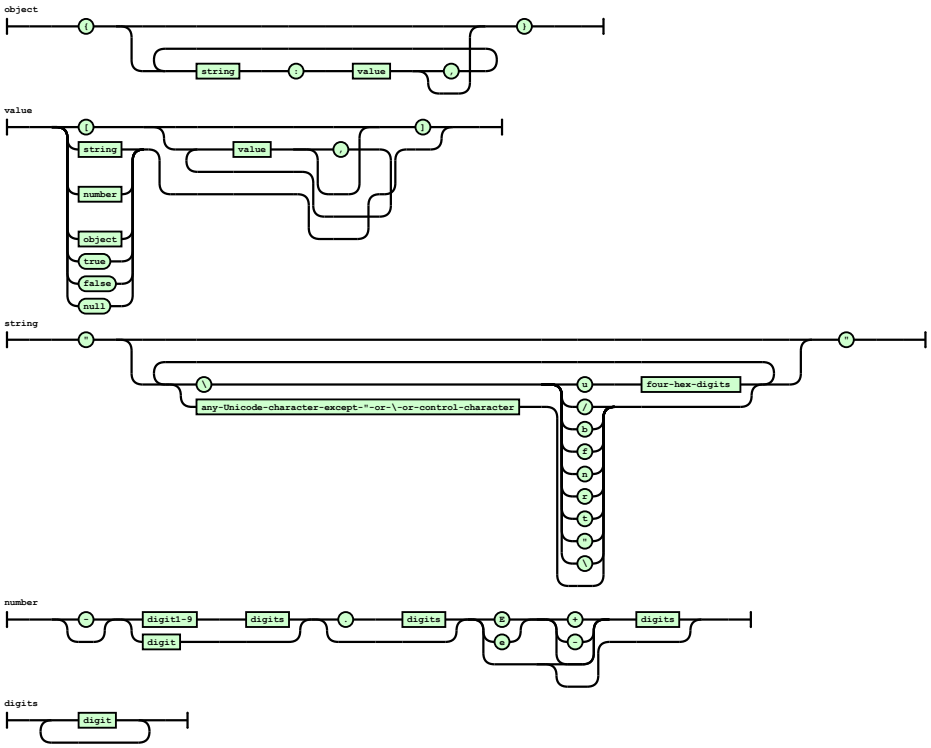

Fig. 11. A syntax diagram for the complete JSON grammar.

References

1. Jensen, K., Wirth, N.: PASCAL User Manual and Report. Springer (1974)
2. Burroughs Corporation: Command and Edit (CANDE) Language Information Manual. (1971)
3. McCarthy, J.: LISP 1.5 programmer's manual. MIT Press (1965)
4. Braz, L.M.: Visual syntax diagrams for programming language statements. SIGDOC Asterisk J. Comput. Doc. **14** (1990) 23–27
5. Bell, S., Gilbert, E.J.: Learning recursion with syntax diagrams. SIGCSE Bull. **6** (1974) 44–45

6. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
7. Crockford, D.: Introducing JSON. `http://json.org` (2015) Accessed: Jun 04, 2015.
8. Atkins, Jr., T., Sapin, S.: CSS Syntax Module Level 3. `http://www.w3.org/TR/css-syntax-3` (2015) Accessed: Jun 04, 2015.
9. Dopler, M., Schörgenhumer, S.: EBNF Visualizer. `http://dotnet.jku.at/applications/Visualizer` (2015) Accessed: Jun 04, 2015.
10. Thiemann, P.: Ebnf2ps: Peter's Syntax Diagram Drawing Tool. `http://www2.informatik.uni-freiburg.de/~thiemann/haskell/ebnf2ps` (2015) Accessed: Jun 04, 2015.
11. Rademacher, G.: Railroad Diagram Generator. `http://bottlecaps.de/rr/ui` (2015) Accessed: Jun 04, 2015.
12. Dickerson, M., Eppstein, D., Goodrich, M.T., Meng, J.Y.: Confluent drawings: Visualizing non-planar diagrams in a planar way. In Liotta, G., ed.: Proc. 11th Int. Symp. Graph Drawing (GD 2003). Volume 2912 of Lect. Notes in Comput. Sci. Springer (2004) 1–12
13. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Trans. Systems Man Cybernet. **11** (1981) 109–125
14. Bastert, O., Matuszewski, C.: Layered drawings of digraphs. In Kaufmann, M., Wagner, D., eds.: Drawing Graphs: Methods and Models. Volume 2025 of Lect. Notes in Comput. Sci. Springer (2001) 87–120
15. Bekos, M.A., Kaufmann, M., Kobourov, S.G., Symvonis, A.: Smooth orthogonal layouts. Journal of Graph Algorithms and Applications **17** (2013) 575–595
16. Eppstein, D., Goodrich, M.T., Meng, J.Y.: Confluent layered drawings. In Pach, J., ed.: Proc. 12th Int. Symp. Graph Drawing (GD 2004). Volume 3383 of Lect. Notes in Comput. Sci. Springer (2005) 184–194
17. Hunt, III, H.B., Rosenkrantz, D.J., Szymanski, T.G.: On the equivalence, containment, and covering problems for the regular and context-free languages. J. Comput. Syst. Sci. **12** (1976) 222–268
18. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: Proc. 5th ACM Symp. on Theory of Computing (STOC '73). (1973) 1–9
19. Gramlich, G., Schnitger, G.: Minimizing nfa's and regular expressions. J. Comput. Syst. Sci. **73** (2007) 908–923
20. Champarnaud, J.M., Coulon, F.: NFA reduction algorithms by means of regular inequalities. Theor. Comput. Sci. **327** (2004) 241–253
21. Han, Y.S., Wood, D.: Obtaining shorter regular expressions from finite-state automata. Theor. Comput. Sci. **370** (2007) 110–120